Sudoku Game and Solver Using Backtracking

Jana Seleem*, Anasalla Eldamanhoury†, Abdelrahman AboIsmaiel‡, Yassin Ebrahim§, Farida Alfaham¶
*Department of Computer Science and Engineering, AUC
900232698

†Department of Computer Science and Engineering, AUC
900232280

‡Department of Computer Science and Engineering, AUC
900232646

§Department of Computer Science and Engineering, AUC
900237672

¶Department of Computer Science and Engineering, AUC
900233775

***Abstract*—This report presents a comprehensive design and implementation of a Sudoku game and solver system, developed using C++ and the Qt framework. The project incorporates several advanced algorithmic techniques from constraint satisfaction problems (CSPs), including recursive backtracking, domain reduction, and graph-based constraint propagation. The application features a user-friendly GUI with real-time interaction, a hint mechanism based on domain analysis, remove-option support, pen-mode note-taking, and a dynamic scoring system adjusted by difficulty level. The system was rigorously tested for correctness, performance, and usability, demonstrating robust capabilities across a range of puzzle complexities.**

I. Introduction

Sudoku is a popular logic puzzle where players fill a 9x9 grid so that each row, column, and 3x3 subgrid contains digits from 1 to 9 exactly once. Although the rules are simple, the search space is combinatorially large, making algorithmic solving both a technical challenge and an academic interest [3]. This project goes beyond solving Sudoku as a mathematical puzzle and reimagines it as a full-featured interactive game.

Our objective was to create a responsive application that supports puzzle solving, player interaction, error feedback, hint generation, and performance tracking. The solver is enhanced by constraint propagation, domain filtering, and custom data structures. The game provides a polished GUI built in Qt, including player actions like input, erasure, pen-mode annotations, difficulty selection, and scoring.

II. Problem Definition

The project tackles two intertwined problems: First, how to solve any valid Sudoku puzzle efficiently while minimizing redundant calculations; and second, how to provide users with an intuitive, responsive interface that makes the experience engaging and educational.

Sudoku is modeled as a CSP where each cell is a variable with a domain (values 1-9). Constraints are that no value may repeat in any row, column, or 3x3 subgrid. The solution must assign values to all cells without violating any constraints [2].

Additional requirements include ensuring real-time GUI responsiveness, dynamic difficulty adjustment, a scoring system tied to performance, and support for user-friendly features like remove and hinting.

III. Methodology and Implementation

A. Recursive Backtracking

Our solver uses depth-first recursive backtracking. It identifies the first unfilled cell and iteratively tries inserting values from 1 to 9. For each attempt, it checks whether the number violates any constraints. If valid, it proceeds to the next cell. If no valid number can be placed, the algorithm backtracks to the previous cell and tries a different value [1].

This basic strategy is enhanced by domain filtering: the algorithm selects the next cell to explore based on heuristic measures (like domain size) to prioritize cells with fewer options, which are more likely to produce conflicts early. This is an application of the Minimum Remaining Values (MRV) [3].

B. Constraint Propagation

Whenever a value is inserted into a cell, constraint propagation eliminates that value from the domains of all its neighbors (cells sharing the same row, column, or 3x3 box). This process is implemented in a custom method that traverses a neighbor list generated using a graph structure.

The propagation ensures early detection of unsolvable paths. If at any point a domain becomes empty, the solver backtracks immediately. This allows the solver to prune invalid branches in the recursion tree early.

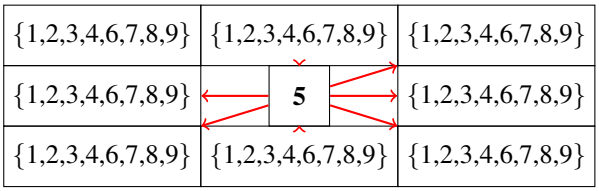

Fig. 1. Constraint propagation: Value 5 assigned to center cell. Arrows show affected cells losing 5 from their domains.

C. Graph-Based Constraint Model

The 81 Sudoku cells are nodes in a graph. Each node is connected via edges to all other nodes in its row, column, or subgrid. This creates a dense graph with up to 20 neighbors per cell.

This model allows efficient lookup of constraints and easy implementation of propagation by traversing neighbors. Each vertex maintains a list of neighbors in a custom vector, ensuring $O(1)$ access during traversal. The graph is built once at initialization and reused throughout the solving process [2]. Our implementation focuses on enforcing Arc Consistency (AC) for propagation [4].

D. Domain Management Using Custom Sets

We implemented our own hash-based 'unorderedSet' to store each cell's domain. During puzzle generation and solving, these sets are updated in response to insertions and removals. This structure allows the hint system to quickly assess domain sizes and the solver to maintain accurate tracking.

E. Puzzle Generation

To ensure uniqueness, puzzle generation begins by filling the three diagonal 3x3 boxes randomly. Since these boxes do not interact, they offer a conflict-free starting point. The rest of the board is filled using the solver.

After a full board is generated, a number of cells are blanked out based on the selected difficulty level (easy: 30–35 clues, hard: 17–22). Each time a cell is removed, the system checks that the board still has a unique solution using a secondary validation solver. This guarantees that generated puzzles are both solvable and non-trivial.

F. Hint System

Hints are based on domain analysis. The system scans all empty cells and selects the one with the smallest domain (i.e., most constrained). It then suggests one value from that domain. This mimics real human-solving techniques.

To prevent abuse, each difficulty level enforces a maximum number of hints. Hint usage affects scoring and is tracked across game sessions.

G. Scoring System and Difficulty Scaling

Each puzzle begins with a base score (e.g., 2000 for easy, 1500 for medium, 1000 for hard). Points are deducted for using hints, wrong inputs, and time taken (1 point per second after a grace period). Final scores are displayed upon completion.

Difficulty level also determines:

- Maximum hints allowed (10 for easy, 5 for medium, 3 for hard)
- Number of removed cells
- Score penalty per hint or error

This encourages strategic play and adds replayability.

H. Graphical User Interface (GUI)

The GUI, developed with Qt, includes:

- An interactive 9x9 grid
- Real-time input validation and formatting
- Pen mode toggle with red annotations
- Hint button with usage counter
- Score and time display

IV. Data Specifications

In our Sudoku system, the core data element is a 9×9 grid representing the puzzle board. Each cell is either a fixed number (clue) or an empty slot awaiting a valid digit from 1 to 9. The initial state of the board is dynamically generated based on the selected difficulty level.

Internally, each cell is stored as a graph vertex that holds:

- A domain set (custom hash-based `unorderedSet`) of possible valid values.
- A list of neighbors (cells sharing row, column, or box).
- A fixed flag indicating whether the cell is part of the original puzzle.

Three main types of boards are used:

1) **Solution Board:** Fully filled and correct grid generated via recursive backtracking and constraint propagation.
2) **Playable Puzzle:** Derived from the solution board by removing a number of values (based on difficulty), ensuring a unique solution.
3) **Player Board:** A live copy of the puzzle where player inputs and hints are applied.

Each difficulty level affects the initial dataset as follows:

- **Easy:** 35–40 prefilled cells.
- **Medium:** 25–30 prefilled cells.
- **Hard:** 17–22 prefilled cells.

Domains are constantly updated during solving and gameplay, meaning the board is not static—it evolves as values are inserted or removed and constraints are propagated.

V. Analysis and Critique

Our Sudoku solver, implemented through recursive backtracking enhanced with constraint propagation, demonstrates correctness and reliability across all difficulty levels. However, its performance and complexity must be evaluated in the broader landscape of existing solving strategies, especially those that utilize advanced metaheuristics or alternative traversal techniques.

Backtracking is a complete algorithm that ensures a solution will be found if it exists. It explores the solution space by recursively assigning values to empty cells, reverting decisions if constraints are violated. The time complexity of pure

backtracking can be approximated as $O(9^m)$, where m is the number of empty cells. However, in our implementation, the effective complexity is improved using the Minimum Remaining Value (MRV) heuristic and constraint propagation, which help prune infeasible branches early. In practice, most easy and medium puzzles are solved in a short time, with harder puzzles taking slightly longer.

VI. Conclusion

This Sudoku solver and interactive game demonstrate the application of CSP techniques in a real-world problem. Through the use of custom-built data structures, recursive algorithms, and a robust GUI, the system achieves both technical efficiency and user-centered design. The solver is reliable and flexible, and the interface encourages learning and strategic thinking. The modular design allows future extensions such as multiplayer competition, leaderboard tracking, or adaptive puzzle generation using machine learning.

Acknowledgment

We would like to thank Dr. Dina Mahmoud and Dr. Ashraf Abdelraouf for their guidance throughout this project. We also appreciate the feedback provided by our peers during testing sessions.

References

[1] M. Thenmozhi, P. Jain, S. Anand, and S. Ram, "Analysis of Sudoku Solving Algorithms," *International Journal of Engineering and Technology (IJET)*, vol. 9, no. 3, pp. 1745–1748, 2017.

[2] S. Chatterjee, S. Paladhi, and R. Chakraborty, "A Comparative Study on the Performance Characteristics of Sudoku Solving Algorithms," *IOSR Journal of Computer Engineering*, vol. 16, no. 5, pp. 69–77, 2014.

[3] P. Berggren and D. Nilsson, "A Study of Sudoku Solving Algorithms," Bachelor's Thesis, Royal Institute of Technology, Sweden, 2012.

[4] I. Howell, R. Woodward, B. Y. Choueiry, and C. Bessiere, "Solving Sudoku with Consistency: A Visual and Interactive Approach," in *Proc. of the 27th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2018, pp. 5829–5831.